Postgres and SQL database

There are different types of databases :-

1. **NoSQL databases**

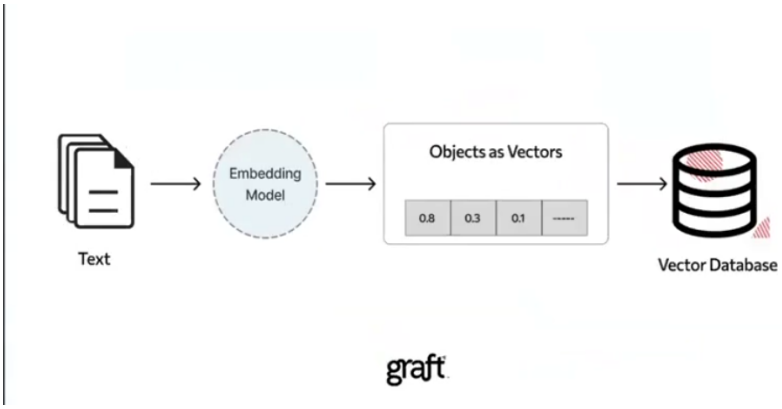
- Store data in a `schema-less` fashion. Extremely lean and fast way to store data.
- Examples - MongoDB, Firebase, etc..

2. **Graph databases**

- Data is stored in the form of a graph. Specially useful in cases **where `relationships` need to be stored (social networks)**
- Examples - Ne04j

3. **Vector Databases**

- Stores data in the form of vectors
- Useful in Machine learning
- Examples - Pinecone



4. **SQL Databases**

- **Stores data in the form of rows** (similar to `collection` in `mongoDB`)
- Most full sack applications will use this
- Examples - MySQL, Postgres, Prometheus, etc..

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

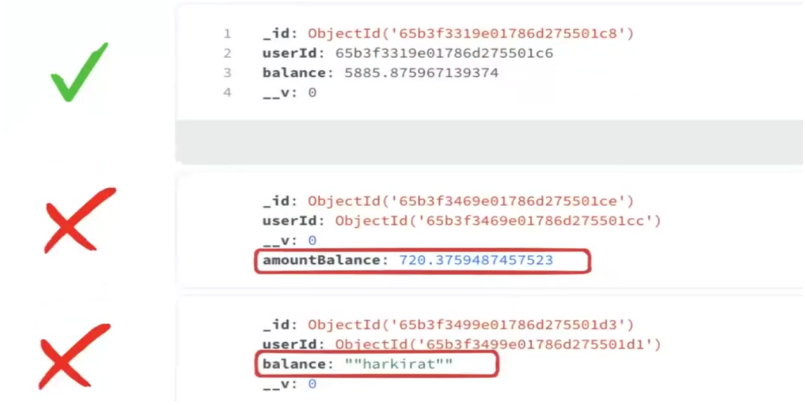
Why Not NoSQL ??

You might've used MongoDB

It's schemaless properties make it ideal to for bootstraping a project fast But as your app grows, this property makes it very easy for data to get `currupted`

💡 What is Schemaless ??

Different rows can have different `schema` (keys / types)



Notice in the above pic, user ne `balance` ki jagah `amountBalance` daal diya then also the `mongoDB` is not complaining, also in the other pic, you can see `amount` me `string` bhej diya(instead of `number`)

Problems ->

1. Can lead to **inconsistent database**
2. Can cause **runtime errors**
3. Is **too flexible for an app that needs strictness**

Upsides or Advantages ->

1. Can **move very fast**
2. Can **change schema very easily**

> 📌 You might think that mongoose does add strictness to the codebase because we used to define a schema there. That strictness is present at the Node.js level, not at the DB level. You can still put in erroneous data in the database that doesn't follow that schema.

Why SQL ??

SQL databases have a strict schema. They require you to :- (**This is also the STEPs to start using the database**)

1. Define your schema
2. Put in data that follows that schema
3. Update the schema as your app changes and perform `migrations`

So there are 4 parts when using an SQL database (not connecting it to Node.js, just running it and putting data in it)

1. Running the database.

2. Using a library that let's you connect and put data in it.
3. Creating a table and defining it's schema .
4. Run queries on the database to interact with the data (Insert/Update/Delete)

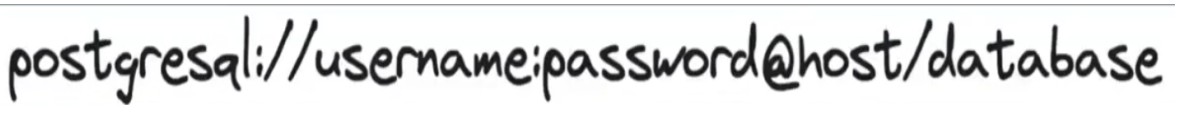
Creating a database

You can start a Potgres database in a few ways -

1. **Using neondb**(easiest way to get `postgres` database) +go to Neon DB [this is decent service that lets you create a server]
    - click on `new project`
    - give the name to the project
    - finally click on `create project`
    - after this you will come to a window where you can find the connection string for postgres (it is similar to that provided and similar to that in `mongoDb`) [Notice the "*" present in the string is where you are going to put the password]
2. **Using docker locally**
    - `docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres`
    - Connection String is below
    - `postgresql://postgres:mysecretpassword@localhost5432/postgres?sslmode=disable`
3. **Using docker on windows**
    - How to run postgrSQL in windows terminal(if you have docker installed).
        - first run docker gui application that help in running commands in terminal.
        - After that run it with the docker instance by the help of following command present after this
        - for the first time if the image is not downloaded use the below command
        - `docker run --name my-postgresl -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres`
        - if the docker image is there, prior to use the it can simply be runned by `docker run <image name>`
        - After that,
        - use `docker exec -It my-postgresl psql -U postgres -d postgres` this command in terminal .
        - then enter the password and it will connect to localhost Postgress instance .
        - now you will be inside the postress command line that looks like `postgres-#`
        - U can check it by running `\\dt`, (the command to display all the tables.)

> 📌 The connection string is similar to the string we had in mongoose.

lets understand about the **url or connection string**



Using a library that lets you connect and put data in it

1. **psql** `psql` is a terminal-based front-end to PostgreSQL. It provides an interactive command-line interface to the PostgreSQL (or TimescaleDB) database. With psql, you can type in queries interactively, issue them to PostgreSQL, and see the query results.

> 💡 **How to connect to your database ?**

`psql` Comes bundled with postgresql. You don't need it for this tutorial. We will directly be communicating with the database from Node.js

```
psql -h p-br0ken-frost-69135494.us-east-2.aws.neon.tech -d database_name_here -U IOOxdevs
```

talking about the above url ->

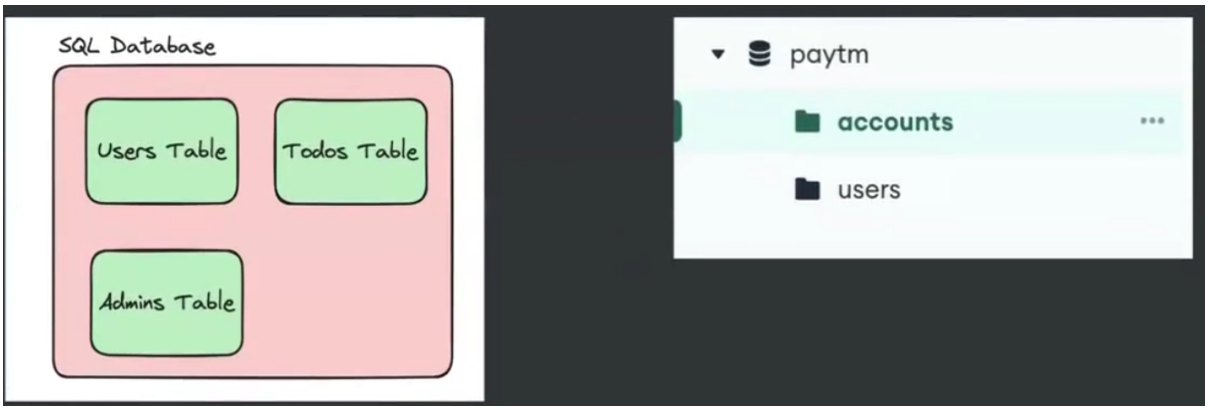
- `-h` -> means where is this hosted ?? you can see this in the connection string generated from `neonDB` after `@`

1. **pg** `pg` is a `Nodejs` library that you can use in your backend app to store data in the Postgres DB (similar to `mongoose`). We will be installing this eventually in our app.

Creating a table and defining its `schema`

Tables in SQL

A single database can have multiple tables inside. Think of them as collections in `MongoDB` database.



> 📌 **In `mongoDB`, it is known as COLLECTIONS and in `MySQL`, it is known as TABLES**

Until now, we have a databypse that we can interact with. The next step in case of postgres is to define the `schema` of your tables.

SQL stands for `Structured query language`. It is a language in which you can describe what/how you want to put data in the database.

To create a table, the command to run is -

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
```

```
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

There are few parts of this SQL statement, lets decode them one by one

1. CREATE TABLE users

`CREATE TABLE users` **->** This command initiates the creation of a new table in the database named `users`

2. id SERIAL PRIMARY KEY

- `id` -> The **name of the first column in the users table**, typically **used as a unique identifier for each row (user)**. Similar to `_id` in mongodb
- `SERIAL` -> A PostgreSQL-specific data type for **creating an auto-incrementing integer.** *Every time a new row is inserted, this value automatically increments, ensuring each user has a unique id .*
- `PRIMARY KEY` : This constraint specifies that the **id column is the primary key for the table, meaning it uniquely identifies each row.** *Values in this column must be unique and not null.*

3. email VARCHAR(255) UNIQUE NOT NULL

- `email` -> The name of the second column, intended to store the user's username.
- `VARCHAR(50)` -> A variable character string data type that can **store up to 50 characters.** It's used here to *limit the length of the username.*
- `UNIQUE` : This constraint ensures that **all values in the `username` column are unique across the table.** *No two users can have the same username.*
- `NOT NULL` : This constraint **prevents null values from being inserted into the `username` column.** *Every row must have a username value.*

4. password VARCHAR(255) UNIQUE NOT NULL

same as above, can be non unique

5. created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP

- `created_at` -> The name of the fifth column, intended to **store the timestamp when the user was created.**
- `TIMESTAMP WITH TIME ZONE` -> This data type **stores both a timestamp and a time zone,** allowing for the precise tracking of when an event occurred, regardless of the user's or server's time zone.
- `DEFAULT CURRENT_TIMESTAMP` -> This default value **automatically sets the created_at column to the date and time at which the row is inserted into the table, using the current timestamp of the database server.**

If you have made the database then try running the above code for creating the table and updating it

Interacting with the database

There are 4 thing you'd like to do with database

CREATE

```
INSERT INTO users (username, email, password)
VALUES ('username_here', 'user@example.com', 'user_password');
```

> 📌 Notice how you didnt have to specify the `id` because it auto increments

UPDATE

```
UPDATE users
SET password = "new_password"
WHERE email = 'user@example.com';
```

You can add more complexity to the above case

```
UPDATE users
SET password = 'new_password', username= '123123'
WHERE email 'harkirat@gmail.com' AND username= ' harkirat' ;
```

The above SQL query will first check the user which has username = 'harkirat' and email = 'harkirat@gmail.com' and then it will update its corresponding password to be 'new_password' and it username to be '123123' from 'harkirat'

DELETE

```
DELETE FROM users
WHERE id = 1;
```

SELECT

used to show the column from the table

```
SELECT * FROM users
WHERE id = 1;
```

* **will print all the column present in the table**

if you want specific column to be selected and shown then

```
SELECT username, email FROM users;
```

this will show all the `username`, `email` present in the `users` table.

How to do queries from a Node.js app ??

In the end, postgres exposes a protocol that someone needs to talk to be able to send these commands (update, delete) to the database.

`psql` is one such library that takes commands from your terminal and sends it over to the database.

To do the same in a Node.js , you can use one of many `Postgres clients` similar in `mongoDB`. for example -> `mongoose` is one of `mongoDB` client

pg library(one of the postgres client)

pg library

Non - blocking(thread is not blocked for the process like `async`, `await` and others...) postgresSQL client for Node.js

Documentation -> https://node-postgres.com/

to use `pg`, after initialising the `node` project with `typescript` enabled write the below command (you know how to do this)

```
npm install pg
```

if using `typescript` then

```
npm install pg @types/pg
```

connecting to the postgres db

```
import { Client } from 'pg'

// as Client is CLASS so to use it first make the instance of the class

const pgClient = new Client("connection_string_to_connect_your_postgres_db") //
Either you do this OR
const client = new Client({
    host: 'my.database-server.com', // present in your connection string, anything
after '@' is a part of the host
```

```
    bort: 5334,
    database: 'database-name',
    user: 'database-user', // present in your connection string
    password: 'secretpassword!!', // present in your connection string
    ssl : true // use this only when inside the connection string, sslmode =
require written
})
// As Client can take any of the 2 arguments (one is the connection STRING and
second is providing all the data in OBJECT form and then connect to the database)

pgClient.connect() // REMEMBER this is asynchronus function as this is the code to
connect to the string (SO IT WILL TAKE TIME) // wrap inside the async function

async function main() {
    await pgClient.connect()
}
```

for example :-

```
import {Client} from "pg"

const pgClient = new Client ("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-
snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

// if above is the connection string then its corresponding OBJECT type will be

const pgClient = new Client({
    user : "neondb_owner",
    password : "wrWG5KI1ziYB",
    port : 5432,
    host : "ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech",
    database : "neondb"
    ssl : true // as our connection string consists of sslmode = require
})
// all of the value has been or can be extracted from the connection string only
except port value
```

Now QUERYING part

about `client.query`

-> whatever the command or sql query you used to give it to any sql database, that query can **be inserted into this function inside '' or "" and then this will execute the command written inside it** [do all the CRUD operations like this]

```
const result = await client.query('SELECT * FROM USERS;')
console.log(result)
```

Write a function to create a users table in your database

```
import {Client} from "pg"

const pgClient = new Client({
    connectionString : "postgresql://your connection string goes here"
})

async function createUsersTable() { // as yahan par server se len den ki baat ho
rhi h so use "await" as much as possible
    await pgClient.connect()
    const result = await pgClient.query(`
        CREATE TABLE users (
            id SERIAL PRIMARY KEY,
            username VARCHAR(50) UNIQUE NOT NULL,
            email VARCHAR(255) UNIQUE NOT NULL,
            password VARCHAR(255) NOT NULL,
            created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
        );
    `)
    console.log(result) // Will give the table as well as whole other things
    console.log(result.rows) // Will give all the entry present in the table
}

createUsersTable();
```

Code for writing doing CRUD operation from the database

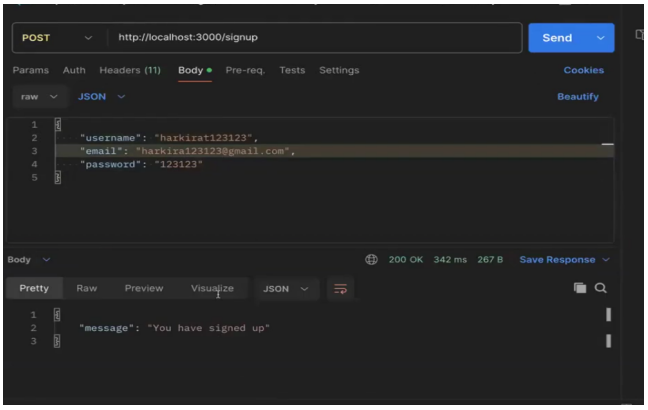
Now how do you actually use it in your website to store the important things in the database. A demo of it is given below:-

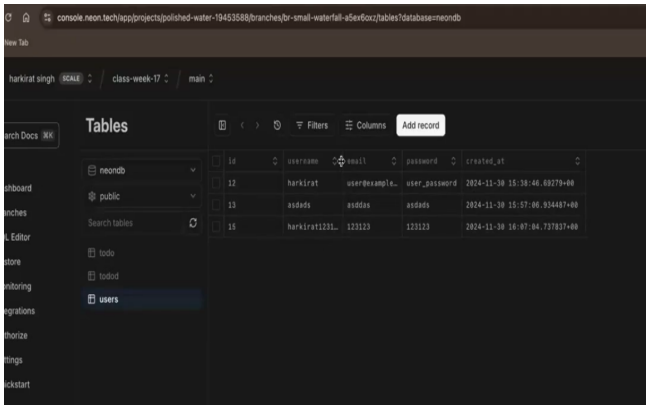
```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

pgClient.connect();

app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    let sqlQuery = "INSERT INTO users (username, email, password) VALUES ("
        sqlQuery += username;
```

```
        sqlQuery += ",";
        sqlQuery += email;
        sqlQuery += ",";
        sqlQuery += password;
        sqlQuery += ");"

    // EITHER WRITE LIKE THE ABOVE or WRITE LIKE THE BELOW (just in one line)
[BOTH will work]

    const insertQuery = `INSERT INTO users (username, email, password) VALUES
('${username}', '${email}', '${password}');` // 2

    const response = await pgClient.query(insertQuery)

    res.json({
        message : "You have signed up"
    })
})

app.listen(3000)
```

> ⚠️ **Remember ->** always put all the variables values inside the `''` or `""` (see the `// 2` line of code) while writing the SQL Query

Ouput of the above code ->





see in the right pic, you got the message "You have signed up" and if you see to the neon db site then under the `users` table you can see the new entry data of what has been sent from the `postman` (see `15` entry of the table)

💡 What are the problems here in the above code ?

1. Backend Crashes -> If you send something which is against the rules defined when creating the table (like username should be unique etc..) the your **Backend crashes**

- to fix it -> **put it inside the `TRY-CATCH` Block**

```
import express from "express"
import { Client} from "pg";
```

```
const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    try {
        const insertQuery = `INSERT INTO users (username, email, password) VALUES
('${username}', '${email}', '${password}');`

        const response = await pgClient.query(insertQuery)

        res.json({
            message : "You have signed up"
        })
    }catch(e){
        res.status(400).json({
            message : "Error while signing up"
        })
    }
})

app.listen(3000)
```

putting it inside the `TRY-CATCH` block will solve the things but you will see that this will still give the ERROR Reason is the 2nd thing to keep in mind

2. SQL Injection

SQL Injection

A term you have widely heard when it comes to hacking any website or steal some data from it.

Now you will be sending all the things like `username`, `password` and `email` inside the browser and eventually it will going to some endpoint. Now this endpoint can be extracted from the developers mode and then through `postman`, you can send the request with all the credentials required (`username`, `password` and `email`). Here `SQL Injection` comes into the picture.

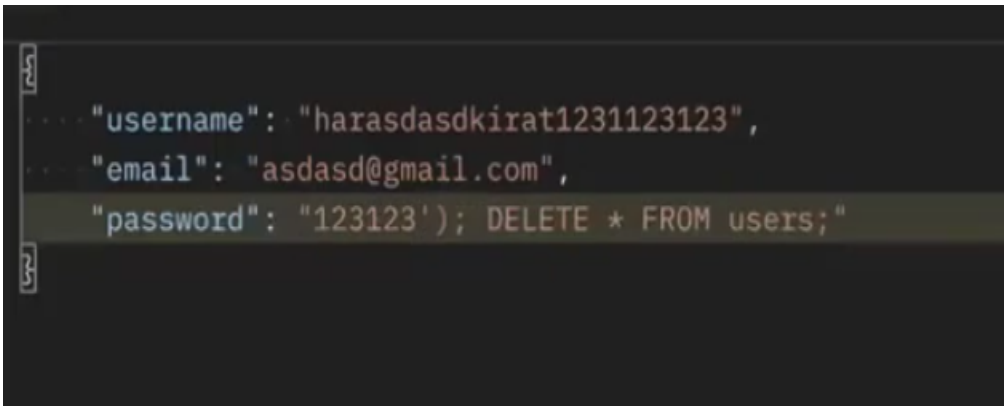
User can send **ABNOXIOUS thing (or bad data) from the** `postman`

something like this :-

so you can say that **I WILL USE `ZOD` to validate the INPUT** and their `types` (with some constraint)

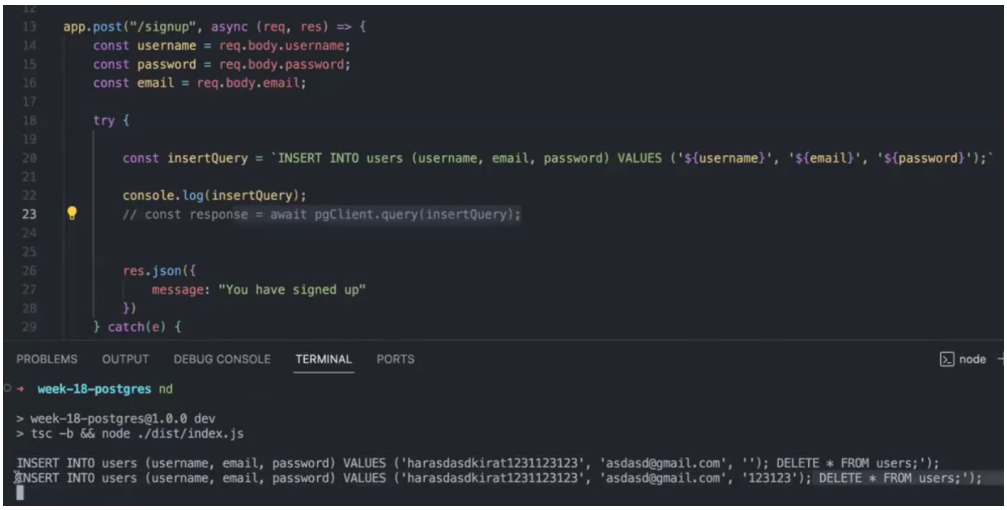
BUT `ZOD` will help you in the case of `NoSQL` database, here it cannot help much as something like `SQl injection` exists here.

Now user has the capabillity to set **any password** so he can **inject `SQL` inside the password and do something which will make your website vulnerable**



User has injected a query to delete all things from the database and with the help of password

if you try to console what is actually being queried to your server and eventually to your backend it looks something like this

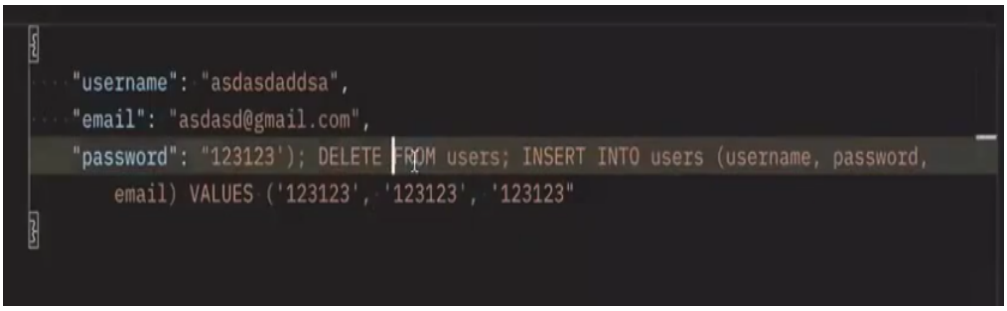


Ignore the first query it was for some other password, look at the second query which prints on console. Now go to the sql editor and paste this query you will see that

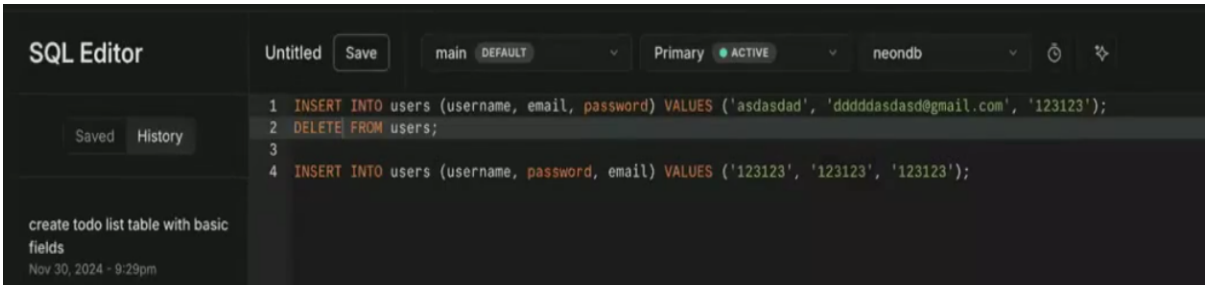
1. users table me corresponding email, username, and password insert ho jayega and then
2. users table se sare records DELETE ho jayega as you have written the second query in that way only

Can you see how much vulnerable our app or website has became ??

Another example (More better way sirf query he h isme nothing extra characters which will lead to run the query without even getting the error)



Now you can see its totally valid (3 queries h isme) will look something like this



Now the above statement will execute like this manner ->

- first it will insert the given data in the users table
- second **it will delete all the things inside the users table**
- third it will insert the data in the users table(and only this data now exists in the users table)

With this power, it can do anything with the database (**CRUD operations basically**) and without even getting notices (**Think how dangerous it is**)

💡 How to fix this ??

-> **Pretty easy actually, you can achieve it by just changing the way you write the query**

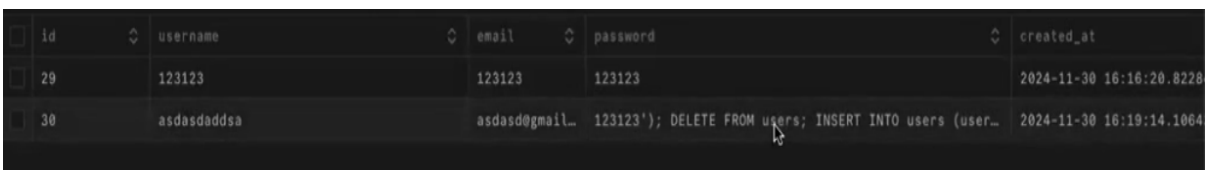
```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
```

```
a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()


app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    try {
        // Instead of writing like this
        // const insertQuery = `INSERT INTO users (username, email, password)
VALUES ('${username}', '${email}', '${password}');`
        // WRITE LIKE THIS
        const insertQuery = `INSERT INTO users (username, email, password) VALUES
($1, $2, $3);`

        // and then adding the value of 1, 2 and 3 inside the response variable
like this (GIVE IT INSIDE THE ARRAY)
        const values = [username, email, password]
        const response = await pgClient.query(insertQuery, values)
        // Benefit of writing like this see // 2

        res.json({
            message : "You have signed up"
        })
    }catch(e){
        res.status(400).json({
            message : "Error while signing up"
        })
    }
})

app.listen(3000)
```

Explantion of `// 2` code By writing like this you have seperated the way data was going inside the sql

`INSERT INTO users (username, email, password) VALUES ($1, $2, $3)` -> ye alg se pahunchega database me and value of `username, email, password` alg se pahunchega. now isse hoga kya ki values of all these 3 will be **STORED NOT APPENDED inside the original string due to which IT GET NEVER RAN**

will look something like this in the database



Notice the sql query **got stored not executed and hence you prevented the `SQL injection`**

In this way, you can also find that someone was trying to inject the `sql` inside the database

💡 **Access token and Refresh token**

-> **Problem with JWTs ->** If your jwt gets leaked then anyone can access your account, there is no way to revoke JWT. so to solve this, you can store the JST in the database and check if it has revoked

BUT it has problem too -> you are hitting the datbase too much and with every request you are checking the database that whether the JWT has been revoked or not

so **ACCESS and REFRESH token works as Middleground** ->

- **REFRESH token are stored inside the database while ACCESS token aren't** so the user send the request with Access token and Refresh token. If their access token is valid for now(usually it is valid for 1 hour) then you let them through and not hit the database but if the access token is not valid then you hit the database, fetch the database, use the refresh token and then again generate a new access token

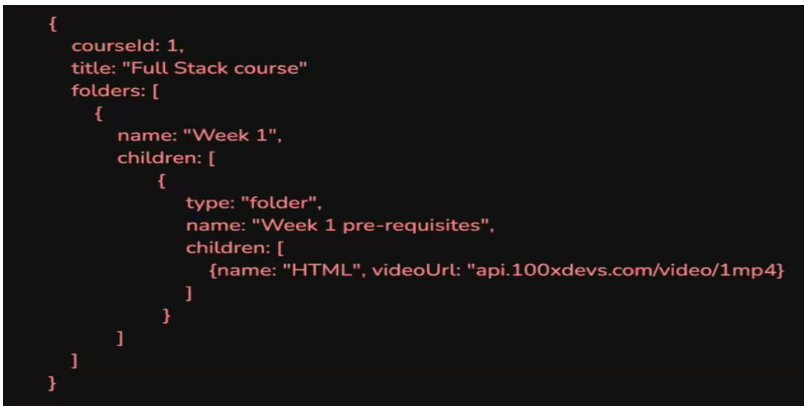
💡 **Why this is great ?**

-> If the access token gets leaked then they only have that for an hour (after an hour, it will become invalid) and if the refresh token gets leaked then they can revoke it as they are stored inside the database.

Relationships and Transactions

if you know this then you can try to build app100x [becomes tough as it has **Nested structure**]

if it would have been using `mongoDb` as the database then the schema would have looked similar to the below image

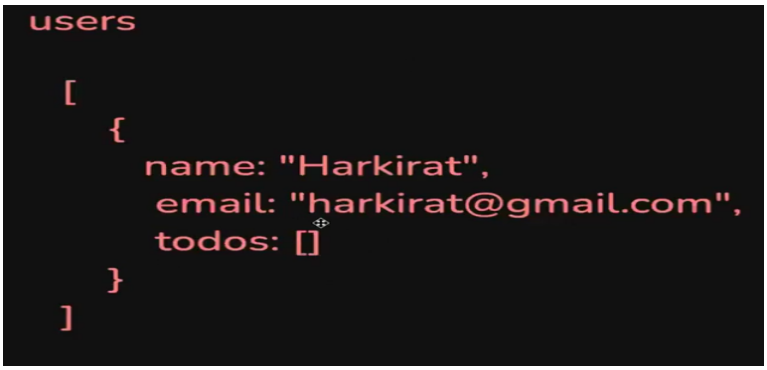


> although for a **nested structure type site, its better to use the `NoSQL` database as you dont know the limit of nesting (it can be infinitely nested i.e. you can add more nesting here)**

This is similar to what we have studied in `mongoDb` relationships

Relationships let you store data in different tables and `relate` it with each other.

Although you can put all the things inside one table in the `NoSQl` database (which is the advantage as well as disadvantage for it) lets understand it by taking the example of `to-do`



you can see you have inserted all the todos inside an array and that is indirectly inside a single table known as `users` table.

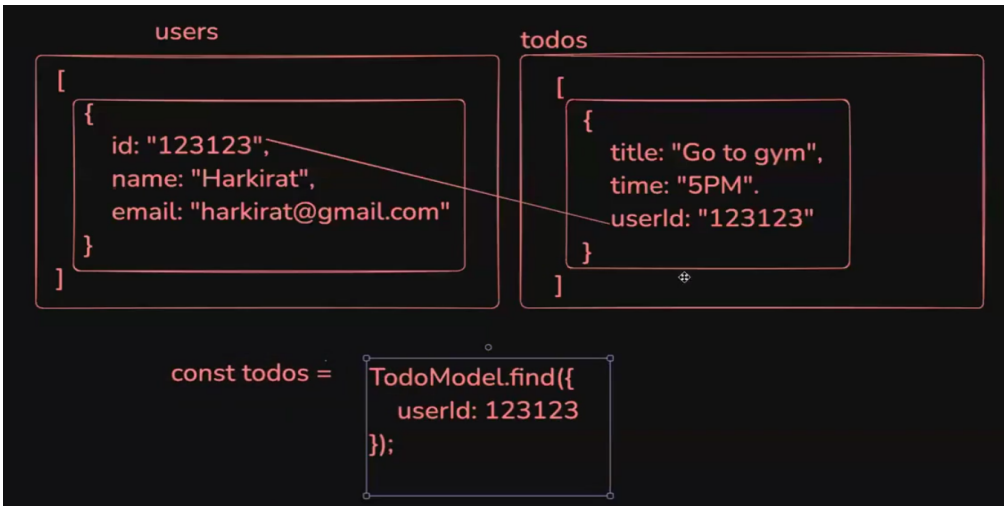
Updside of this ->

- the schema is easier to write and read
- sb kuch ek jagah pr h

Downside of this ->

- any user first has to go inside the `users` table to access `todos` which is not good

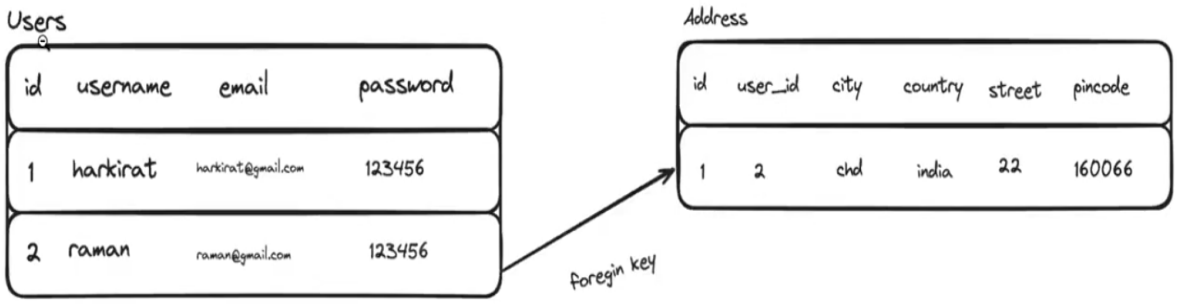
hence **Decomposing data(seperating them) is good thing** so you should seperate both `todos` and `users` table and then **try to relate them** something like the below



Notice the **Relationship denoted by RED line (userId related to id)** and hence we were able to find out the user `todos` with the help of the id and **relationship which has been created**

> 📌 **If you are seperating tables, then you must ensure that the RELATIONSHIPS should exist among the tables created**

so **relationships in `SQl` database looks like this ->**



This is called a `relationship`, which means that the `Address` table is related to the `Users` table.

When defining the table, you need to define the `relationships`

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE addresses(
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    city VARCHAR(100) NOT NULL,
    country VARCHAR(100) NOT NULL,
    street VARCHAR(255) NOT NULL,
    pincode VARCHAR(20),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE // at last you
have to write the relationship
    // This means that addresses table ka jo user_id field h that REFERENCES to
the id of the users table
    // ON DELETE CASCADE -> agar kbhi users table me kuch delete hua to jahan
jahan uska relationship exist krta h wahan wahan pr(here addresses table) me jake
v delete kr do
    // similar to this is
    // ON DELETE RESTRICT -> this will restrict you to delete FIRST all the
relationship which exists for the main table and then only you can delete an entry
in the main table
);
```

SQL Query

To insert the addrerss of a user ->

```
INSERT INTO addresses (user_id, city, country, street pincode)
VALUES (1, 'New York', 'USA', '123 Broadway St', '10001');
```

Now if you want to get the address of a user given an `id`, you can run the following query

```
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = 1;
```

Lets try to define all the above things inside and call it from the `node.js` app

```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()


app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    const city = req.body.city
    const country = req.body.country
    const street  = req.body.street
    const pincode = req.body.pincode


    try {
        // Making the users table and then inserting into it
        const insertQuery = `INSERT INTO users (username, email, password) VALUES ($1, $2, $3) RETURNING id;`
        // RETURNING id means -> id of the new user which has just now been inserted will be returned by writing this line of code
        // id which got returned from the above line, store it inside the userId
        const userId = response.rows[0].id

        const values = [username, email, password]
        const response = await pgClient.query(insertQuery, values) // 2

        // Making the address table and then inserting into it
        const addressInsertQuery = `INSERT INTO addresses (city, country, street,
```

```
    pincode, user_id) VALUES ($1, $2, $3, $4, $5);`

            const addressInsertResponse = await pgClient.query(addressInsertQuery,
    [city, country, street, pincode, userId])


            res.json({
                message : "You have signed up"
            })
        }catch(e){
            res.status(400).json({
                message : "Error while signing up"
            })
        }
    })

    app.listen(3000)
```

The above code has some issues, can you see this ??

-> see the `// 2` line of code, lets suppose there is case that after execution of `// 2` code, server crashed or lets say database me time lg gya iske baad(next wale line ko execute krne me), in this case, `users` table to ban gya data v chla gya but `addresses` table to hold pe chla gya na.

Now for a website, this is one **of the critical issue you will often face**

A more practical scenario of this case will be `Paytm`

```
app.post("/signup", async(req, res) => {
    const query1 = "UPDATE USER WHERE user_id = 1 SET balance = balance - 10" // 1
    const query1 = "UPDATE USER WHERE user_id = 2 SET balance = balance + 10" // 2
ek se kat ke dusre me jayega
})
```

Now here if the above case will come then ek query to excute ho gya but dusra nhi hua (**means `// 1` execute ho gya but `// 2` nhi**), this will totally create the chaos among the users.

This is where `Transaction` comes into the picture.

Transaction in Postgres SQL

Whenever you have 2 or more queries which are interrelated to each other very strictly (means ek ke change hone se dusre me change hona JARURI he h), in this case YOU SHOULD WRAP the queries inside the TRANSACTION

Transactions in SQL

💡 Good question to have at this point is what queries are run when the user signs up and sends both their information and their address in a single request.

Do we send two SQL queries into the database? What if one of the queries (address query for example) fails? This would require `transactions` in SQL to ensure either both the user information and address goes in, or neither does

```
BEGIN; <-- Start transaction

INSERT INTO users (username, email, password)
VALUES ('john_doe','john_doe@example.com','securepassword123');

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (currval('users_id_seq'),'New York', 'USA', '123 Broadway St', '10001');

COMMIT; <-- End transaction
```

you have basically wrapped both the `Insert` code inside the TRANSACTION

Applying the above concept to above `paytm` example ->

```
app.post("/signup", async(req, res) => {
    const query3 = "BEGIN;"
    const query1 = "UPDATE USER WHERE user_id = 1 SET balance = balance - 10"  //
1
    const query1 = "UPDATE USER WHERE user_id = 2 SET balance = balance + 10" // 2
    const query4 = "COMMIT;"
})
```

just by wrapping inside the `BEGIN` and `COMMIT` you have said that **all the queries which are present inside this block, EITHER THEY ALL WILL EXECUTE OR NONE OF THEM WILL EXECUTE (at a particular point of time). Remember `A`(Atomicity) in `ACID` property of database**

simply saying, if `// 1` has executed and `// 2` failed, then `// 1` will **Revert back**

wrapping the above code(example of `node.js` site) inside the transaction

```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()
```

```
app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    const city = req.body.city
    const country = req.body.country
    const street  = req.body.street
    const pincode = req.body.pincode


    try {

        const insertQuery = `INSERT INTO users (username, email, password) VALUES
($1, $2, $3) RETURNING id;`
        const addressInsertQuery = `INSERT INTO addresses (city, country, street,
pincode, user_id) VALUES ($1, $2, $3, $4, $5);` // 1

        // wrapped inside the transaction
        await pgClient.query("BEGIN;")
        const values = [username, email, password]
        const response = await pgClient.query(insertQuery, values)
        const userId = response.rows[0].id // 2
        const addressInsertResponse = await pgClient.query(addressInsertQuery,
[city, country, street, pincode, userId])
        await pgClient.query("COMMIT;")


        res.json({
            message : "You have signed up"
        })
    }catch(e){
        res.status(400).json({
            message : "Error while signing up"
        })
    }
})

app.listen(3000)
```

💡 **You can have question in your mind that if the server crashed at `// 1` line of code, then `// 2` will be able to get `id` as `RETURNING id` to execute ho gya but `// 2` to chla he nhi so in this case it tries to create the new entry and eventually fails**

Joins

Lets try to solve a problem

💡 given the id = 12, return me all the data(means user related info. + address related info.) related to user whose id = 12 if someone hits the backend point (`/metadata`) ??

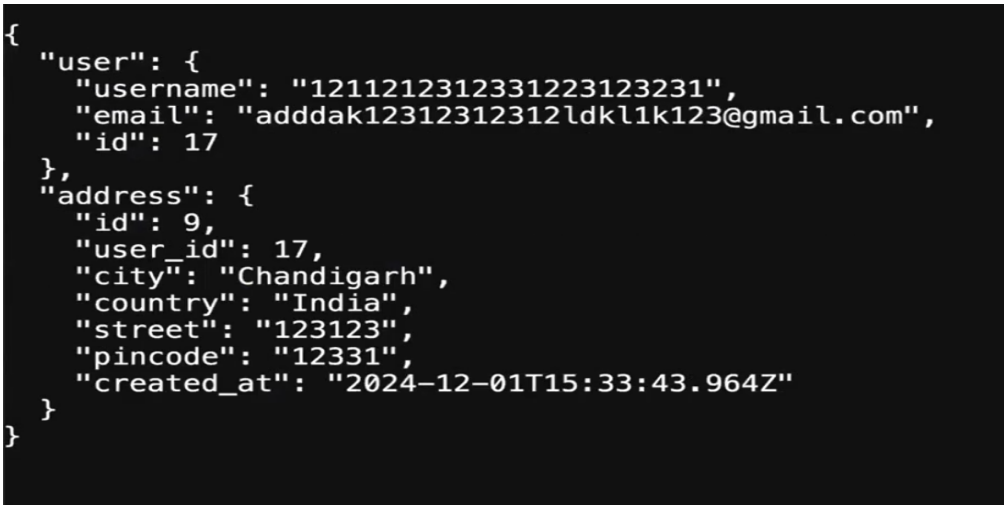
-> how will you do the above problem

```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    // signup logic already written above

})

app.get("/metadata", async(req, res) => {
    const id = req.body.id // given this id = 12 here how to get all the data

    // const query1 = await `SELECT * FROM users WHERE id = $1` // make sure you do not add "*" as it will return all the data (including some sensitive data such as password) so instead jitna jarurat h utna he return kro
    const query1 = await `SELECT username, email, id  FROM users WHERE id = $1`
    const response1 = pgClient.query(query1, [id]);

    const query2 = await `SELECT * FROM addresses WHERE user_id = $1`
    const response2 = pgClient.query(query2, [id]);

    res.json({
        user : response1.rows[0]
        address : response2.rows[0] // 2
        address : response2.rows // 3
```

```
    })
  })
  app.listen(3000)
```

when you will go to link -> https://localhost:3000/metadata?id=12"

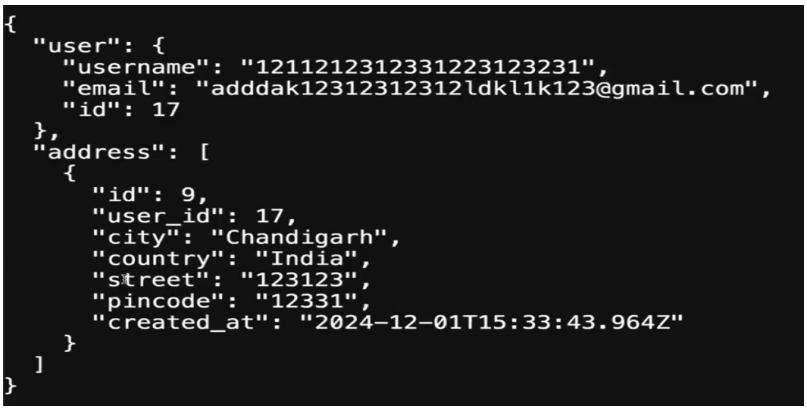
you will see the output



You can see all the data related to user with id = 12 in the above picture

But the above code has some error, **ideally user has more than one address, so it should be able to see all the address stored and linked to it in the database and hence you should RETURN AN ARRAY of addresses**

so instead of returning only one row, it should return **all the rows** so write `// 3` line of code instead of `// 2`. (if a new entry will come that will also now will be reflected now)



Now the output looks like this -> you have got the **ARRAY of address being returned**

💡 why this approach is bad ??

-> due to the reason that you are **sending two seperate queries**[if some change happened after the 1st query so 2nd query will not be knowing about this (later it will know but for the time being, it will not be able to know about this), and hence anomalies will occur]

The Better version of the above approach is `JOINS`

JOINS

Defining relationships is easy.

What's hard is Joining data from two (or more) tables together.

For example, if I ask you to fetch me a users details and their address, what SQL would you run?

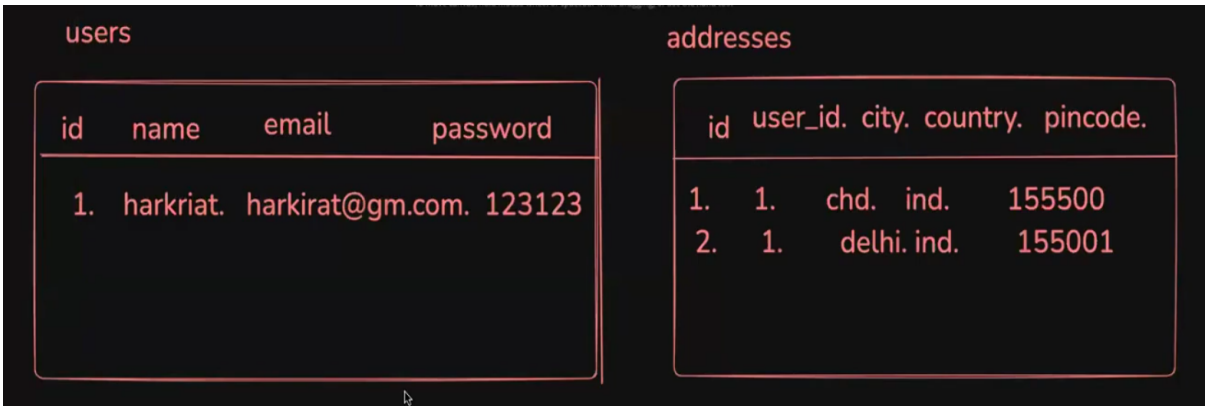
Approach 1 -> Bad one

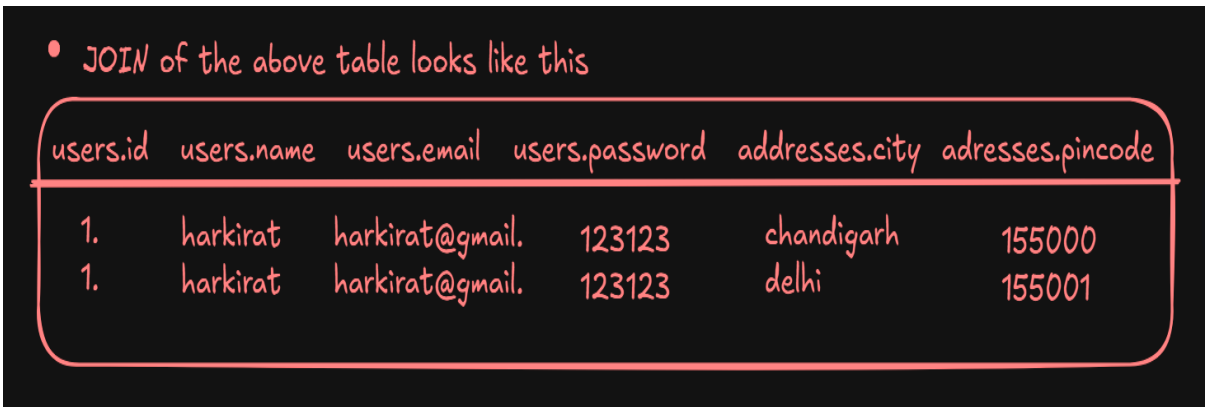
```
- Query 1: Fetch user's details

SELECT id, username, email
FROM users
WHERE id = YOUR_USER_ID

- Query 2 Fetch user's address
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = YOUR_USER_ID
```

Approach 2 -> using joins

```
SELECT users.id, users.username, users.email, addresses.city, addresses.country,
addresses.street, addresses.pincode
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE usersid = `1`;

// OR you can write the above code like this also(just used ALIAS in this nothing
extra)
SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pincode
FROM users u
JOIN addresses a ON u.id = a.user_id
WHERE u.id = YOUR_USER_ID
```

> 📌 **Remember ->** **while using `JOIN`, a new table is created.**





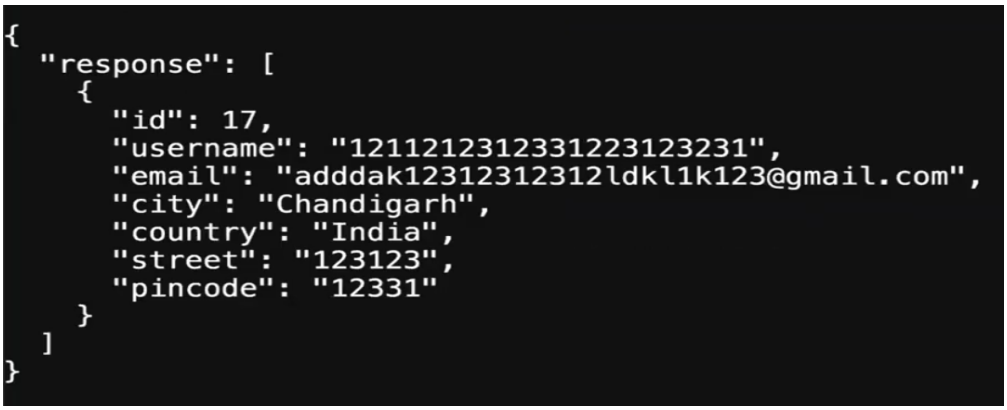
so applying the above concept in solving or making the endpoint -> `/metadata`

the code looks like this ->

```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    // signup logic already written above

})

app.get("/metadata", async(req, res) => {
    const id = req.query.id
    const query = `SELECT users.id, users.username, users.email, addresses.city,
addresses.country, addresses.street, addresses.pincode FROM users JOIN addresses
ON users.id = addresses.user_id WHERE usersid = $1;`
    // Instead of $1 why not this -> ${id} REASON -> SQL Injection
```

```
    const response = await pgClient.query(query, [id])
    res.json({
        response : response.rows // REMEMBER -> if you will do simply send
"response" then PURA OBJECT de dega(consisting of hell lot of things (from SQL
command which involved to other SQl realted entity value))
        // by writing "response.rows" -> ye bas table me jo pda h wo dega (NOTHING
EXTRA)
    })
})
app.listen(3000)
```

Now the output looks like this ->



> 📌**Remember -> `JOIN` is a expensive task to perform**[computationally tough task to perform]
>
> > lets say one table has M (lets say 10000) rows and other has N(lets say 35) rows then to join both the table **in WORST CASE would be M*N (10000 * 35)**

Other than the above demerit, it(JOIN) has mostly advantages :-

1. **Reduced Latency**
2. **Simplified Application Logic**
3. **Transactional Integrity**

Types of JOINS

INNER JOIN

Returns rows when there is AT LEAST one match in both tables. If there is no match, the rows are not returned. It's the **most common type** of join.

Use Case :- Find All Users With Their Addresses. If a user hasn't filled their address, that user shouldn't be returned.

> 📌**Remember -> BY DEFAULT -> `JOIN` works the same way as `INNER JOIN`**

```
SELECT users.username, addresses.city, addresses.country,
addresses.street,addresses.pincode
FROM users
INNER JOIN addresses ON users.id = addresses.user_id;
```

LEFT JOIN

Returns ALL rows from the LEFT table, and the MATCHED rows from the RIGHT table.

Use case :- To **list all users from your database along with their address information** (if they've provided it), you'd use a LEFT JOIN. Users without an address will still appear in your query result, but the address fields will be NULL for them.

```
SELECT users.username, addresses.city, addresses.country, addresses.street,
addresses.pincode
FROM users
LEFT JOIN addresses ON users.id = addresses.user_id;
```

RIGHT JOIN

Returns ALL rows from the RIGHT table, and the MATCHED rows from the LEFT table.

simply saying -> **if there is something which is in left and not in right, then that should come as well as vice versa of this should also come and those which are present in both left and right that should also come**

> 📍 **RIGHT JOIN = !(LEFT JOIN)**

Use case :- Given the structure of the database, a **RIGHT JOIN would be less common since the** `addresses` **table is unlikely to have entries not linked to a user due to the foreign key constraint.** However, if you had a situation where you start with the `addresses` table and optionally include user information, this would be the theoretical use case.

```
SELECT users.username, addresses.city, addresses.country, addresses.street
addresses.pincode
FROM users
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

FULL JOIN

Returns rows when there is a MATCH in one of the tables. It effectively combines the results of both LEFT JOIN and RIGHT JOIN.

> 📍 **FULL JOIN = RIGHT JOIN + LEFT JOIN**

Use case :- A FULL JOIN would **combine all records from both users and addresses showing the relationship where it exists.** Given the constraints, this might not be as relevant because every `address` should be linked to a `user`, but if there were somehow orphaned records on either side, this query would reveal them.

```
SELECT users.username, addresses.city, addresses.country, addresses.street,
addresses.pincode
FROM users
FULL JOIN addresses ON users.id = addresses.user_id;
```